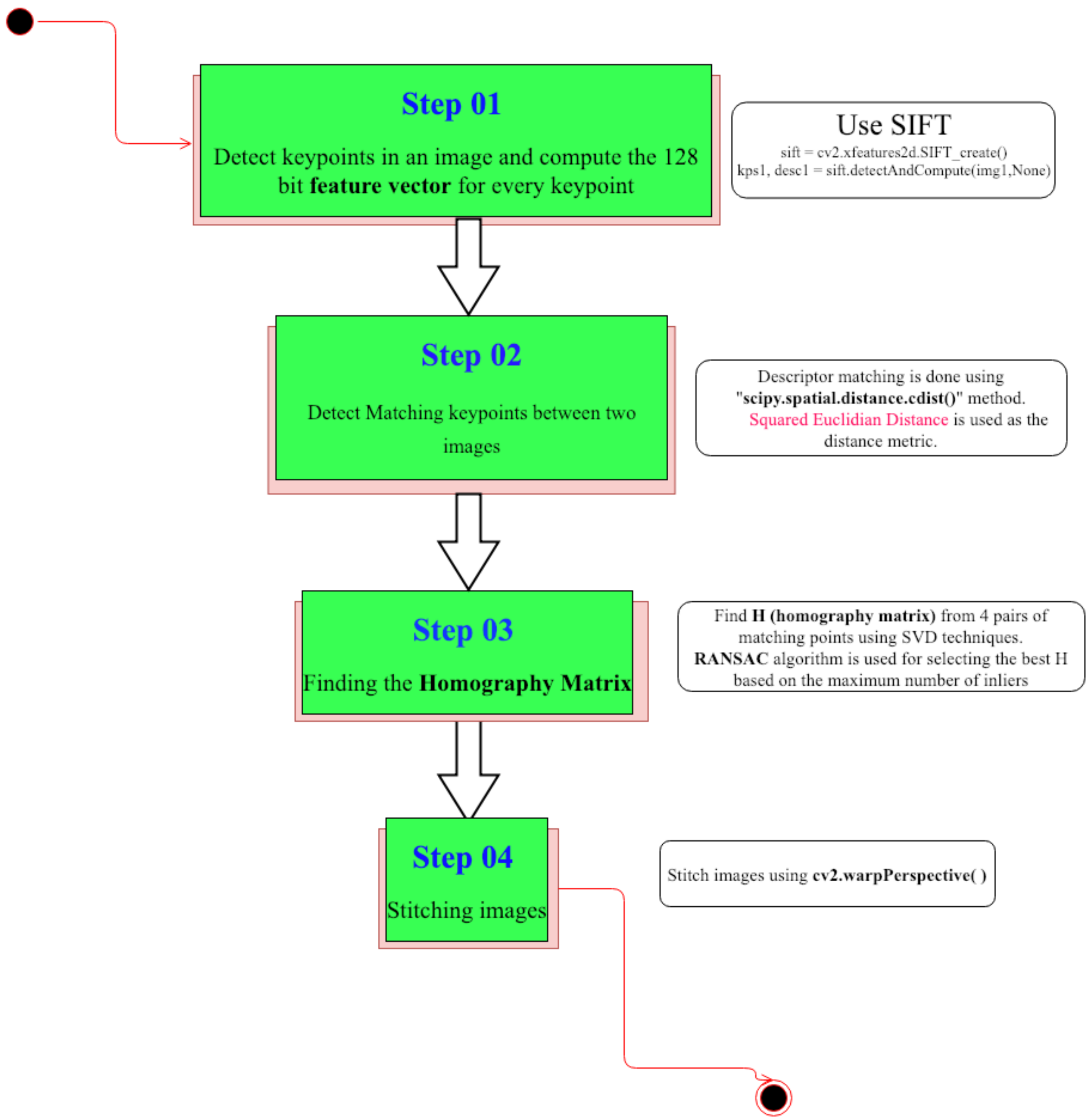
CSE 573 : UB ID : **50291708**

Image Stitching

Stitch at most 3 images to create a panorama

Flow chart of my workflow :

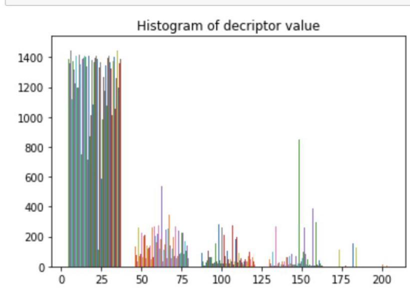


[Step 01 :]

I use **SIFT** method to detect the **key points** (points which are scale-invariant and rotation-invariant) in an image. This method also computes the **descriptor** for each point. The descriptor is a feature vector of 128 bit which we can use for matching.

```
In [5]: sift = cv2.xfeatures2d.SIFT_create()
        kps1, desc1 = sift.detectAndCompute(img1,None) # Detects keypoints and computes the descriptors
        kps2, desc2 = sift.detectAndCompute(img2,None)
        # Here kps1 is a list of keypoints and desc1 is a numpy array of shape "Number_of_Keypoints × 128".

In [6]: type(kps1)
Out[6]: list

In [7]: len(kps1)
Out[7]: 1509

In [8]: type(kps1[0])
Out[8]: cv2.KeyPoint

In [9]: kps1[0].pt  # Get the coordinates of a keypoint
Out[9]: (4.403946876525879, 1337.380859375)

In [10]: type(kps1[0].pt)
Out[10]: tuple

In [11]: type(desc1)
Out[11]: numpy.ndarray
```

```
In [17]: plt.hist(desc1, bins=5)
         plt.title("Histogram of decriptor value")
         plt.show()
```



Step 02 :

After detecting the keypoints and descriptors of two images I calculate the matching keypoints using **Squared Euclidian Distance** as the distance metric. I also tried the hamming distance but I was getting better result with squared euclidian distance as I figure out the proper threshold. I use the threshold value 7000.

Each feature point I obtain using SIFT on an image is usually associated with a 128-dimensional vector that acts as a descriptor for that specific feature. The SIFT algorithm ensures that these descriptors are mostly invariant to in-plane rotation, illumination and position. So if a feature from one image is to be matched with the corresponding feature in another image, their descriptor needs to be matched to find the closest matching feature. This can be done in various ways, but I choose squared euclidean distance between these descriptors.

```
n [2]: def matching_keypoints_sqeuclidian(kps1, kps2, desc1, desc2):
           """
               Find matching descriptors and corresponding keypoints between 2 images.
               Descriptor matching is done using "scipy.spatial.distance.cdist()" method.
               Squared Euclidian Distance is used as the distance metric.
           """
           pairwiseDistances = cdist(desc1, desc2, 'sqeuclidean')
           threshold    = 7000

           # Return a list of 2 elements : 1st element contains all row numbers and 2nd element contain all col numbers
           points_Row_Col = np.where(pairwiseDistances < threshold)

           points_in_img1    = points_Row_Col[0]  # Row numbers represent points in the 1st image
           points_in_img2    = points_Row_Col[1]  # Col numbers represent points in the 2nd image

           # List of tuples as each coordinate (x,y) is a tuple
           coordinates_in_img1 = []
           coordinates_in_img2 = []

           for point in points_in_img1:
               coordinates_in_img1.append(kps1[point].pt)

           for point in points_in_img2:
               coordinates_in_img2.append(kps2[point].pt)

           return np.concatenate( ( np.array(coordinates_in_img1),np.array(coordinates_in_img2) ), axis=1)
```

```
In [ ]: def matching_keypoints_hamming(kps1, kps2, desc1, desc2):
            """
                Find matching descriptors and corresponding keypoins between 2 images.
                Descriptor matching is done using "scipy.spatial.distance.cdist()" method.
                Squared Euclidian Distance is used as the distance metric.

            """
            pairwiseDistances = cdist(desc1, desc2, 'hamming')  # normalized hamming distance
            print(np.max(pairwiseDistances))

            # https://stackoverflow.com/questions/22857398/matching-orb-features-with-a-threshold
            threshold    = 0.4

            # return a list of 2 elements : 1st element contains all row number and 2nd element contain all col number
            points_Row_Col = np.where(pairwiseDistances < threshold)

            points_in_img1    = points_Row_Col[0]  # row numbers represent points in the 1st image
            points_in_img2    = points_Row_Col[1]  # col numbers represent points in the 2nd image

            # List of tuples as each coordinate (x,y) is a tuple
            coordinates_in_img1 = []
            coordinates_in_img2 = []

            for point in points_in_img1:
                coordinates_in_img1.append(kps1[point].pt)

            for point in points_in_img2:
                coordinates_in_img2.append(kps2[point].pt)

            return np.concatenate( ( np.array(coordinates_in_img1),np.array(coordinates_in_img2) ), axis=1)
```

Step 03 :

After getting the matching descriptors and corresponding keypoint between two images I calculate the homography matrix (H) which can transform the first image to the second image. I used **RANSAC** algorithm with parameter 1000 (totale iteration) and 0.5 (threshold for error) for calculating the best H. In each iteration I choose 4 points randomly and calculate the H and then count the total number of inliers. I chosee the H which gives the maximum inlier count.

```
n [3]: def get_homography(fourMatchingPairs):
           """
               Find H (homography matrix) from the input (4 pairs of matching points)
           """
           # Solving Ah = 0
           A = []

           for matchingPair in fourMatchingPairs:
               # Point_01
               x1 = matchingPair[0]
               y1 = matchingPair[1]

               # Corresponding point of Point_01
               x1_prime = matchingPair[2]
               y1_prime = matchingPair[3]

               '''
               References :
                - https://cseweb.ucsd.edu/classes/wi07/cse252a/homography_estimation/homography_estimation.pdf
                - http://laid.delanover.com/homography-estimation-explanation-and-python-implementation/
                - https://math.stackexchange.com/questions/494238/how-to-compute-homography-matrix-h-from-corresponding-points
               '''
               a1 = [-x1, -y1, -1,   0,    0,  0, x1*x1_prime, y1*x1_prime, 1*x1_prime ]
               a2 = [  0,   0,  0 , -x1, -y1, -1, x1*y1_prime, y1*y1_prime, 1*y1_prime ]

               A.append(a1)
               A.append(a2)

           # converting the list to numpy ndarray
           A = np.array(A)

           # using SVD technique
           U, sigma, V_transpose = np.linalg.svd(A)

           # Last column of V or Last row of V_transpose represents the elements of H
           H = V_transpose[-1].reshape(3,3)
           H = H / H[2,2]   # Since DOF is 8 so the last element of H should be 1

           return H
```

In [4]:
```python
def ransac_algo(matchingPoints,totalIteration):

    # Ransac parameters
    highest_inlier_count = 0
    best_H = []

    # Loop parameters
    counter = 0

    while counter < totalIteration:

        counter = counter + 1

        # Select 4 points randomly
        secure_random  = rand.SystemRandom()

        matachingPair1 = secure_random.choice(matchingPoints)
        matachingPair2 = secure_random.choice(matchingPoints)
        matachingPair3 = secure_random.choice(matchingPoints)
        matachingPair4 = secure_random.choice(matchingPoints)

        fourMatchingPairs=np.concatenate(([matachingPair1],[matachingPair2],[matachingPair3],[matachingPair4]),axis=0)

        # Finding homography matrix for this 4 matching pairs

        # H = get_homography(fourMatchingPairs)
        points_in_image_1 = np.float32(fourMatchingPairs[:,0:2])
        points_in_image_2 = np.float32(fourMatchingPairs[:,2:4])

        H = cv2.getPerspectiveTransform(points_in_image_1, points_in_image_2)

        rank_H = np.linalg.matrix_rank(H)

        # Avoid degenrate H
        if rank_H < 3:
            continue

        # Calculate error for each point using the current homographic matrix H
        total_points = len(matchingPoints)

        points_img1 = np.concatenate( (matchingPoints[:, 0:2], np.ones((total_points, 1))), axis=1)
        points_img2 = matchingPoints[:, 2:4]

        correspondingPoints = np.zeros((total_points, 2))

        for i in range(total_points):
            t = np.matmul(H, points_img1[i])
            correspondingPoints[i] = (t/t[2])[0:2]

        error_for_every_point = np.linalg.norm(points_img2 - correspondingPoints, axis=1) ** 2

        inlier_indices = np.where(error_for_every_point < 0.5)[0]
        inliers        = matchingPoints[inlier_indices]

        curr_inlier_count = len(inliers)

        if curr_inlier_count > highest_inlier_count:
            highest_inlier_count = curr_inlier_count
            best_H = H.copy()

    return best_H
```

Step 04 :

I use cv2.warpPerspective ( ) method to transform the first images according to H.

```python
# Stitching image 1 and image 2
H12 = ransac_algo(matching_keypoints_sqeuclidian(kps1,kps2, desc1, desc2), 1000)

result = cv2.warpPerspective(colorImages[0], H12 ,
                            ( int(colorImages[0].shape[1]*0.8 + colorImages[1].shape[1]*0.8),
                              int(colorImages[0].shape[0]*0.8 + colorImages[1].shape[0]*0.8) )
                           )

result[0:colorImages[1].shape[0], 0:colorImages[1].shape[1]] = colorImages[1]

cv2.imwrite( dir + '/panoroma12.jpg', result)
```

To make the stitching order independent I first find out the center image out of the given 3 images. The image which has the highest matching points with other images is the center image. Then I do forward and backward stitching and check the total number of black pixels to figure out which will give the best panoram.

```
# SIFT feature detection
sift = cv2.xfeatures2d.SIFT_create()

kps1, desc1 = sift.detectAndCompute(images[0],None)
kps2, desc2 = sift.detectAndCompute(images[1],None)
kps3, desc3 = sift.detectAndCompute(images[2],None)

a12 = matching_keypoints_sqeuclidian(kps1,kps2,desc1,desc2)
a13 = matching_keypoints_sqeuclidian(kps1,kps3,desc1,desc3)
a23 = matching_keypoints_sqeuclidian(kps2,kps3,desc2,desc3)

totalMatch_img1 = len(a12) + len(a13)
totalMatch_img2 = len(a12) + len(a23)
totalMatch_img3 = len(a13) + len(a23)

if totalMatch_img1 >= totalMatch_img2 and totalMatch_img1  >= totalMatch_img3:
    centerIdx = 0
elif totalMatch_img2 >= totalMatch_img1 and totalMatch_img2 >= totalMatch_img3:
    centerIdx = 1
else:
    centerIdx = 2

if centerIdx==0:
    # swap 1st and 2nd images
    temp = images[0]
    images[0] = images[1]
    images[1] = temp

    tempC = colorImages[0]
    colorImages[0] = colorImages[1]
    colorImages[1] = tempC

elif centerIdx==2:
     # swap 2nd and 3rd images
    temp = images[2]
    images[2] = images[1]
    images[1] = temp

    tempC = colorImages[2]
    colorImages[2] = colorImages[1]
    colorImages[1] = tempC
else:
    pass

# For new ordering
kps1, desc1 = sift.detectAndCompute(images[0],None)
kps2, desc2 = sift.detectAndCompute(images[1],None)
kps3, desc3 = sift.detectAndCompute(images[2],None)
```

My output Panorama :

UBDATA :